Figure 1
Modified SSA-conversion process

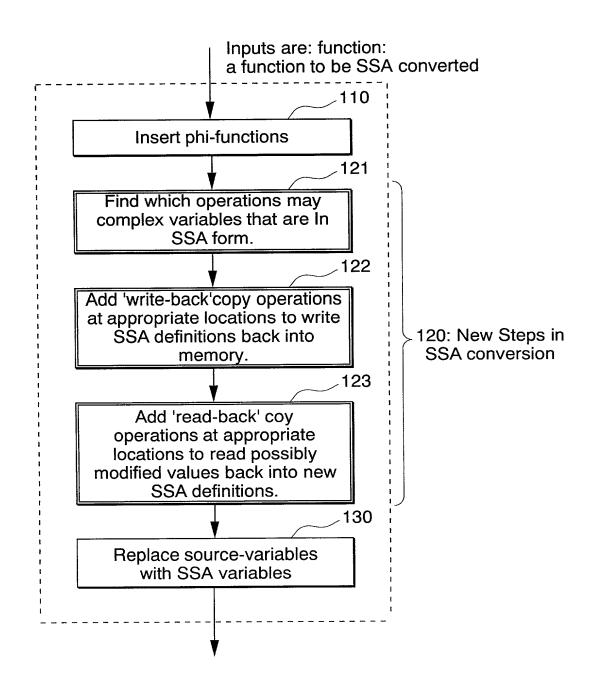


Figure 2
Overall compiler control flow

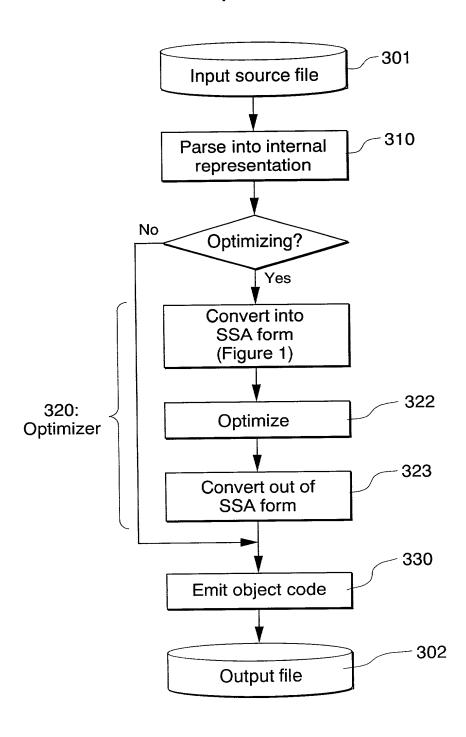


Figure 3
Program representation

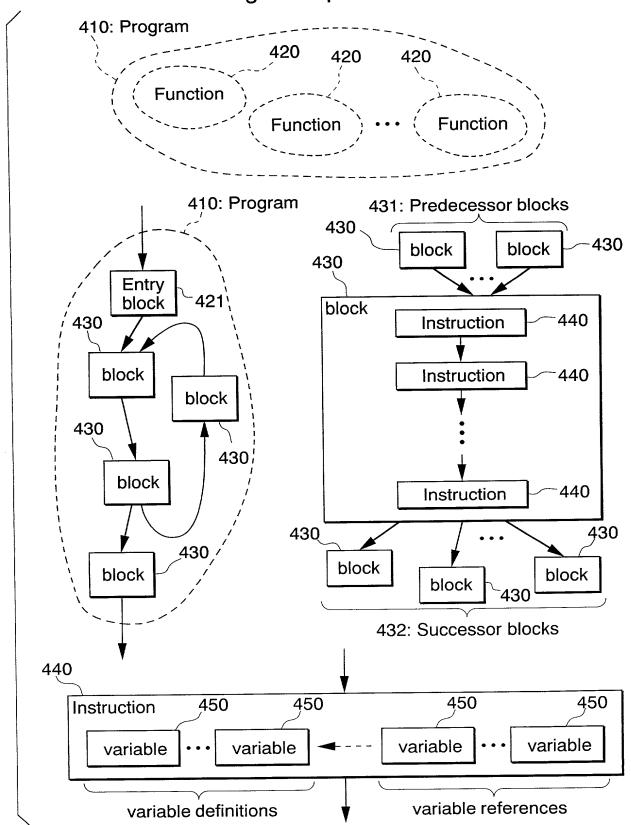
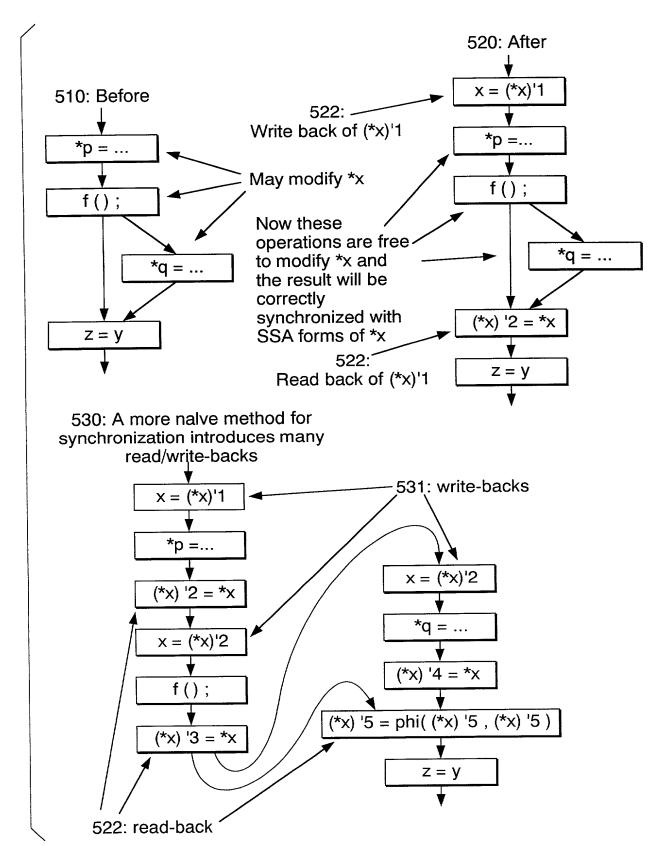
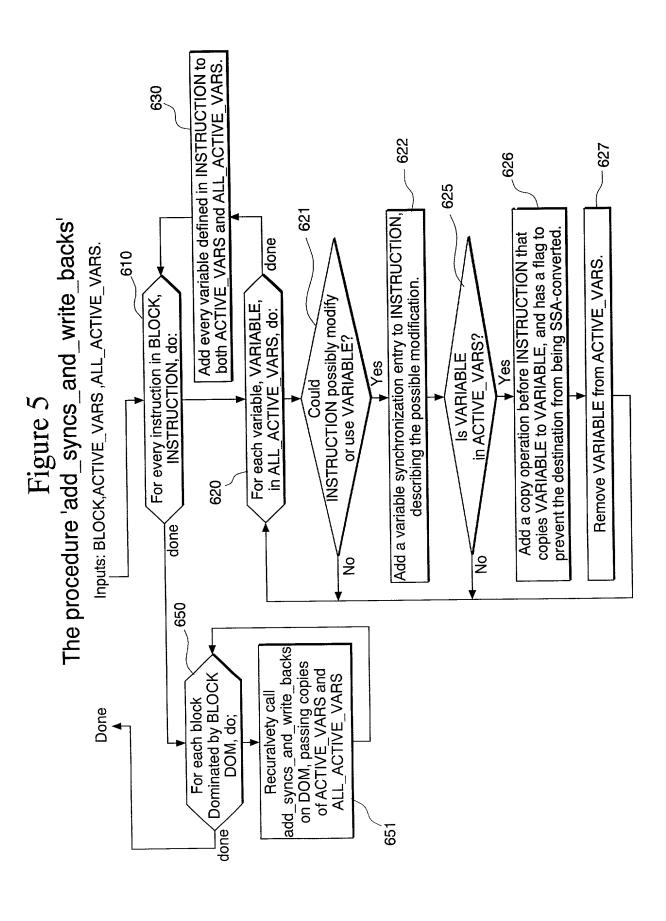


Figure 4
Placement of read/write-backs for the SSA form of *x, (*x)'1





read-back-point, add them are necessary because of If any new phi-functions a new definition at the 720 -711 730 702 701 710 Conversion step (a'.III), insertion of read-backs 731 721 For each read-back, RB, that has been marked as 'used', do; Invoke the procedure 'propagate block read backs' on the head of the queue, after removing its from the queue 741 Initialize the mappings BLOCK BEGIN READ BACK and BLOCK END READ BACKS to empty mappings. marks the source of the copy so that it doesn't have SSA variable conversion performed on it. While PENDING_BLOCKS is not empty, do: Add a copy operation at the read-back-point that copies RB's variable to itself, but also Initialize the queue PENDING BLOCKS to contain only the function's entry block. being the, beginning of the block that RB represents the merge of. Define the 'read-back-point' as Are all of RB's sources also Is RB a 'merge' read-back? marked as 'used'? Inputs: a function Figure 6 ▼ Yes 2 Define the 'read-back-point' as being just after the operation that caused RB to be created done 2 done 742

732

Figure 7A

810 801 822 821 the BLOCK END READ BACKS value of all of BLOCK's predecessor blocks. Where two or more different read-backs for the same variable are present, a 'merge read-back' is created to 820 END_READ_BACKS, assigning the results to OLD_END_BEGIN_READ_BACKS and OLD_END_READ_BACKS respectively, and Look up BLOCK in BLOCK_BEGIN_READ_BACKS and BLOCK Define NEW_BEGIN_READ_BACKS to be the intersection of Inputs: BLOCK, ACTIVE_VARS, ALL_ACTIVE_VARS BACKS, initialized from NEW_BEGIN_READ BACKS for BLOCK in BLOCK BEGIN READ BACKS using an empty set where BLOCK has no entry Make NEW_BEGIN_READ_BACKS the entry Define a new read-back set, NEW_END_READ combine them, staring at BLOCK BEGIN READ BACKS, or is this READ BACKS different from OLD the first time BLOCK has been IS NEW BEGIN processed? Yes **⊳**(৪) The procedure 'propagate_read_backs' Done 870 871 Make NEW_END_READ_BACKS the entry for BLOCK in BLOCK_END_READ_BACKS Add SUCC to PENDING BLOCKS in BLOCK's successor list, do: For each successor, SUCC, END_READ_BACKS and OLD_END_BACKS different? Are NEW 880

done For each variable sync in INSTRUCTION that notes 855 a variable, VARIABLE as possibly written, do: Add a new read-back entry for VARIABLE to NEW_END_READ_BACKS, replacing any existing entry for VARIABLE 860 For each variable definition, VDEF, have an entry in NEW_END_READ_BACKS? in BLOCK, do: Does VDEF Yes 8 The procedure 'propagate_read_backs' Remove the read-back from NEW_END_ READ_BACKS 865 860 Figure 7B done 856 done 845 846 For each operation INSTRUCTION, have an entry in NEW_END_READ______________BACKS? 840 For each variable reference, VREF, 830 Mark that read-back in INSTRUCTION, do: in BLOCK, do: Does VREF Yes as 'used' Remove the read-back from NEW_END_READ_BACKS_ done 847

block6: i (); block7:

p := 5; block8:

return *p;

Figure 8 Example source program

This short C program is used to illustrate the invention:

```
extern int g () , h ( ) , i ( ) , x;
 int foo (int *p)
                                                                                 [810]
    (*p) ++; if (*P > 10)
      {
         g();
         h();
         if (x > 5)
           g ();
         if (x > 3)
           i();
         else
            X = *p;
         *P = 5;
       }
    return *p;
Here's the same program converted to a slightly more primitive form:
  int foo (int *p)
  block1:
                                                                                  [B20]
     p := p + 1;
     if (^{*}P <= 10)
       goto block8;
  block2:
     g();
     h();
     if (x <= 5)
       goto block4;
   block3:
     g();
   block4:
   if (x > 3)
     goto block6;
   block5:
                                                                                  [B40]
     x := *p;
     goto block7;
```

[830]

Figure 9

SSA converted program, with simple implementation of read-backs :

The following is psuedo-C, augmented with the phi' operation, where

```
RESULT = phi (block1: VAL1, ..., blockN:VALN)
```

means `assign VAL1 to RESULT if control-flow comes from block1', and similarly so on for each value of N.

The extra variables 'pvN", where N is an integer, are SSA versions of *P, and are in fact local variables, not dereferences of p.

```
1nt foo (int *p)
  int pvl, pv2, pv3, pv4, pv5, pv6;
block1;
  pvl = *P + 1;
  if (pvl \ll 10)
    goto block8;
block2:
  *P = pvI;
                    /* This writes-back PV1 to *P. */
  g ();
  pv2 = *P:
                     /* This reads-back *P into PV2. */
  *P = pv2;
                     /* This writes-back PV2 to *P. */
  h();
  pv3 = *P:
                     /* This reads-back *P into PV3 */
                                                                              [912]
  if (x <= 5)
    goto block4;
block3:
                     /* This writes-back PV4 to *p, */
  p = pv3;
  g();
                                                                              [911]
                    /* This reads-back *p into PV4. */
  pv4=*p;
block4:
  pv5 = phi (block3: pv4, block2: pv3)
                                                                              [910]
  if (x > 3)
    goto block6;
block5:
  gota block7;
block6;
  i();
block7:
  x = phi (block6: x, block5: pv5);
  pv6 = phi (block1: pv1, block7: 5);
                   /* This writes-back PV6 to *P. */
  *P = pv6;
  return pv6;
}
```

Figure 10

SSA converted program, with the implementation of read-backs described in this patent

```
int foo (int *p)
  int pv1, pv2, pv3;
block1:
  pv1 = *p +1;
                                                                          [1011]
  if (pvl \ll 10)
    goto block8;
block2;
                    /* This writes-back pv1 to *P. */
  p = pv1;
  g();
                                                                          [1021]
  h();
                                                                          [1022]
  if (x <= 5)
    goto block4;
block3;
                                                                          [1023]
  g ();
block4:
                     /* This reads-back *p into pv2, */
  pv2 = *p;
                                                                          [1030]
  if (x > 3)
    goto block6;
block5;
  goto block7;
block6:
  i();
                                                                          [1024]
black7:
                                                                          [1031]
  x = phi (block6 : x, block5 : pv2);
block8:
                                                                          [1010]
  pv3 = phi (block1: pv1, block7: 5);
  P = pv3;
                      /*This writes-back PV3 to *P */
  return pv3;
}
```

}

Figure 11 Register-alloced and SSA-unconverted program

using BBA-form requires having a good register allocator that will merge variables where possible, as it tends to generate a lot of variables with short lifetimes. We assume that here.

```
int foo (int *p)
  int pv;
block1;
  pv = *p + 1;
  if (pv <= 10)
    goto block8;
block2:
                    /* This writes-back pv to *P. */
  'P = pv;
  g();
  h();
  if (x <= 5)
    goto block4;
block3:
  g();
block4:
  if (x > 3)
    goto block6;
block5:
  x=*p;
    goto block7;
block6:
  i ();
block7:
   pv = 5;
block8:
                   /* This writes-back PV to *P. */
  *P= pv
return pv;
```

Figure 12
Original SSA-conversion process

